

Runtime Optimizations for Prediction with Tree-Based Models

Nima Asadi^{1,2}, Jimmy Lin^{1,2,3}, Arjen P. de Vries⁴

¹Dept. of Computer Science, ²Institute for Advanced Computer Studies, ³The iSchool
University of Maryland, College Park

⁴Centrum Wiskunde and Informatica (CWI), Amsterdam

nima@cs.umd.edu, jimmylin@umd.edu, arjen@acm.org

ABSTRACT

Tree-based models have proven to be an effective solution for web ranking as well as other problems in diverse domains. This paper focuses on optimizing the runtime performance of applying such models to make predictions, given an already-trained model. Although exceedingly simple conceptually, most implementations of tree-based models do not efficiently utilize modern superscalar processor architectures. By laying out data structures in memory in a more cache-conscious fashion, removing branches from the execution flow using a technique called predication, and micro-batching predictions using a technique called vectorization, we are able to better exploit modern processor architectures and significantly improve the speed of tree-based models over hard-coded if-else blocks. Our work represents the first instance of an *architecture-conscious* runtime implementation of tree-based models that we are aware of.

1. INTRODUCTION

Recent studies have shown that machine-learned tree-based models, combined with ensemble techniques, are highly effective for building web ranking algorithms [5, 8, 21] within the “learning to rank” framework [14]. Beyond document retrieval, tree-based models have also proven effective for tackling problems in diverse domains such as online advertising [16], medical diagnosis [9], genomic analysis [19], and computer vision [7]. This paper focuses on runtime optimizations of tree-based models that take advantage of modern processor architectures: we assume that a model has already been trained, and now we wish to make predictions on new data as fast as possible. Although exceedingly simple, tree-based models do not efficiently utilize modern processor architectures due to the prodigious amount of branches and non-local memory references in standard implementations. By laying out data structures in memory in a more cache-conscious fashion, removing branches from the execution flow using a technique called predication, and micro-batching predictions using a technique called vectorization, we are able to better exploit modern processor architectures and significantly improve the speed of tree-based models over hard-coded if-else blocks.

Our experimental results are measured in nanoseconds for individual trees and microseconds for complete ensembles. A natural starting question is: do such low-level optimizations actually matter? Does shaving microseconds off an algorithm have substantive impact on a real-world task? We

argue that the answer is *yes*, with two different motivating examples: First, in our primary application of learning to rank for web search, prediction by tree-based models forms the inner loop of a search engine. Since commercial search engines receive billions of queries per day, improving this tight inner loop (executed, perhaps, trillions of times) can have a noticeable effect on the bottom line. Fast prediction translates into fewer servers for the same query load, reducing datacenter footprint, electricity and cooling costs, etc. Second, in the domain of financial engineering, every nanosecond counts in high frequency trading. Orders on NASDAQ are fulfilled in less than 40 microseconds.¹ Firms fight over the length of cables due to speed-of-light propagation delays, both within an individual datacenter and across oceans [12].² Thus, for machine learning in financial engineering, models that shave even a few microseconds off in terms of prediction present an edge.

We view our work as having the following contributions: First, we introduce the problem of *architecture-conscious* implementations of machine learning algorithms to the information retrieval and data mining communities. Although similar work has long existed in the database community [1, 17, 18, 24, 3], to our knowledge, this is the first application of architecture-conscious optimizations for machine learning applications *at runtime* (as opposed to during training). Second, we propose novel implementations of tree-based models that are highly-tuned to modern processor architectures, taking advantage of cache hierarchies and superscalar processors. Third, we illustrate our techniques in a standard, widely-accepted, learning-to-rank task and show significant performance improvements over standard implementations and hard-coded if-else blocks.

2. BACKGROUND AND RELATED WORK

We begin with an overview of modern processor architectures and recap advances over the past few decades. The broadest trend is perhaps the multi-core revolution [15]: the relentless march of Moore’s Law continues to increase the number of transistors on a chip exponentially, but experts widely agree that we are long past the point of diminishing returns in extracting instruction-level parallelism in hardware. Instead, adding more cores appears to be a better use of increased transistor density. Since prediction is an embar-

¹<http://www.nasdaqtrader.com/Trader.aspx?id=colo>

²<http://spectrum.ieee.org/computing/it/financial-trading-at-the-speed-of-light>

rassingly parallel problem, our techniques can ride the wave of increasing core counts.

A less-discussed, but just as important trend over the past two decades is the so-called “memory wall” [3], where increases in processor speed have far outpaced improvements in memory latency. This means that RAM is becoming slower relative to the CPU. In the 1980s, memory latencies were on the order of a few clock cycles; today, it could be several hundred clock cycles. To hide this latency, computer architects have introduced hierarchical cache memories: a typical server today will have L1, L2, and L3 caches between the processor and main memory. Cache architectures are built on the assumption of reference locality—that at any given time, the processor repeatedly accesses only a (relatively) small amount of data, and these fit into cache. The fraction of memory accesses that can be fulfilled directly from the cache is called the *cache hit rate*, and data not found in cache is said to cause a *cache miss*. Cache misses cascade down the hierarchy—if a datum is not found in L1, the processor tries to look for it in L2, then in L3, and finally in main memory (paying an increasing latency cost each level down).

Managing cache content is a complex challenge, but there are two main principles that are relevant to a software developer. First, caches are organized into cache lines (typically 64 bytes), which is the smallest unit of transfer between cache levels. That is, when a program accesses a particular memory location, the entire cache line is brought into (L1) cache. This means that subsequent references to nearby memory locations are very fast, i.e., a cache hit. Therefore, in software it is worthwhile to organize data structures to take advantage of this fact. Second, if a program accesses memory in a predictable sequential pattern (called striding), the processor will prefetch memory blocks and move them into cache, before the program has explicitly requested the memory locations (and in certain architectures, it is possible to explicitly control prefetch in software). There is, of course, much more complexity beyond this short description; see [10] for an overview.

The database community has explored in depth the consequences of modern processor architectures for relational query processing [1, 17, 18, 24, 3]. In contrast, these issues are underexplored for information retrieval and data mining applications. To our knowledge, ours is the first attempt at developing architectural-conscious runtime implementations of machine learning algorithms. Researchers have explored scaling the *training* of tree-based models to massive datasets [16, 20], which is of course an important problem, but orthogonal to the issue we tackle here: given a trained model, how do we make predictions quickly?

Another salient property of modern CPUs is pipelining, where instruction execution is split between several stages (modern processors have between one to two dozen stages). At each clock cycle, all instructions “in flight” advance one stage in the pipeline; new instructions enter the pipeline and instructions that leave the pipeline are “retired”. Pipeline stages allow faster clock rates since there is less to do per stage. Modern *superscalar* CPUs add the ability to dispatch multiple instructions per clock cycle (and out of order) provided that they are independent.

Pipelining suffers from two dangers, known as “hazards” in VLSI design terminology. *Data hazards* occur when one instruction requires the result of another (that is, a data

dependency). This happens frequently when dereferencing pointers, where we must first compute the memory location to access. Subsequent instructions cannot proceed until we actually know what memory location we are accessing—the processor simply stalls waiting for the result (unless there are other independent instructions that can be executed). *Control hazards* are instruction dependencies introduced by if-then clauses (which compile to conditional jumps in assembly). To cope with this, modern processors use *branch prediction techniques*—in short, trying to predict which code path will be taken. However, if the guess is not correct, the processor must “undo” the instructions that occurred after the branch point (“flushing” the pipeline).

The impact of data and control hazards can be substantial: an influential paper in 1999 concluded that in commercial RDBMSes at the time, almost half of the execution time is spent on stalls [1].³ Which is “worse”, data or control hazards? Not surprisingly, the answer is, it depends. However, with a technique called predication [2, 13], which we explore in our work, it is possible to convert control dependencies into data dependencies (see Section 3). Whether predication is worthwhile, and under what circumstances, remains an empirical question.

Another optimization that we adopt, called vectorization, was pioneered by database researchers [4, 24]: the basic idea is that instead of processing a tuple at a time, a relational query engine should process a “vector” (i.e., batch) of tuples at a time to take advantage of pipelining.⁴ Our work represents the first application of vectorization to optimizing machine learning algorithms that we are aware of.

Beyond processor architectures, the other area of relevant work is the vast literature on learning to rank [14], application of machine learning techniques to document ranking in search. Our work uses gradient-boosted regression trees (GBRTs) [5, 21, 8], a state-of-the-art ensemble method. The focus of most learning-to-rank research is on learning effective models, without considering efficiency, although there is an emerging thread of work that attempts to better balance both factors [22, 23]. In contrast, we focus exclusively on runtime ranking performance, assuming a model that has already been trained (by other means).

3. TREE IMPLEMENTATIONS

In this section we describe various implementations of tree-based models, starting from two baselines and progressively introducing architecture-conscious optimizations. We focus on an individual tree, the runtime execution of which involves checking a predicate in an interior node, following the left or right branch depending on the result of the predicate, and repeating until a leaf node is reached. We assume that the predicate at each node involves a feature and a threshold: if the feature value is less than the threshold, the left branch is taken; otherwise, the right branch is taken. Of course, trees with greater branching factors and more complex predicate checks can be converted into an equivalent binary tree, so our formulation is general. Note that our

³Of course, this was before the community was aware of the issue, and so systems have become much more efficient since then.

⁴Note that this sense of vectorization is distinct from, but related to, explicit SIMD instructions that are available in many processor architectures today. Vectorization increases the opportunities for optimizing compilers to generate specialized SIMD instructions automatically.

discussion is agnostic with respect to the predictor at the leaf node, be it a boolean (in the classification case), a real (in the regression case), or even an embedded sub-model.

We assume that the input feature vector is densely-packed in a floating-point array (as opposed to a sparse, map-based representation). This means that checking the predicate at each tree node is simply an array access, based on a unique consecutively-numbered id associated with each feature.

OBJECT: As a high-flexibility baseline, we consider an implementation of trees with nodes and associated left and right pointers in C++. Each tree node is represented by an object, and contains the feature id to be examined as well as the decision threshold. For convenience, we refer to this as the **OBJECT** implementation. In our mind, this represents the most obvious implementation of tree-based models that a software engineer would come up with—and thus serves as a good point of comparison.

This implementation has two advantages: simplicity and flexibility. However, we have no control over the physical layout of the tree nodes in memory, and hence no guarantee that the data structures exhibit good reference locality. Prediction with this implementation essentially boils down to pointer chasing across the heap: when following either the left or the right pointer to the next tree node, the processor is likely to be stalled by a cache miss.

CODEGEN: As a high-performance baseline, we consider statically-generated if-else blocks. That is, a code generator takes a tree model and directly generates C code, which is then compiled and used to make predictions. For convenience, this is referred to as the **CODEGEN** implementation. This represents the most obvious performance optimization that a software engineering would come up with, and thus serves as another good point for performance comparison.

We expect this approach to be fast. The entire model is statically specified; machine instructions are expected to be relatively compact and will fit into the instruction cache, thus exhibiting good reference locality. Furthermore, we leverage decades of compiler optimizations that have been built into GCC. Note that this eliminates data dependencies completely by converting them all into control dependencies.

The downside, however, is that this approach is inflexible. The development cycle now requires more steps: after training the model, we need to run the code generation, compile the resulting code, and then link against the rest of the system. This may be a worthwhile tradeoff for a production system, but from the view of rapid experimentation and iteration, the approach is a bit awkward.

STRUCT: The **OBJECT** approach has two downsides: poor memory layout (i.e., no reference locality and hence cache misses) and inefficient memory utilization (due to object overhead). To address the second point, the solution is fairly obvious: get rid of C++ and drop down to C to avoid the object overhead. We can implement each node as a **struct** in C (comprising feature id, threshold, left and right pointers). We construct a tree by allocating memory for each node (`malloc`) and assigning the pointers appropriately. Prediction with this implementation remains an exercise in pointer chasing, but now across more memory-efficient data structures. We refer to this as the **STRUCT** implementation.

STRUCT⁺: An improvement over the **STRUCT** implementation is to physically manage the memory layout ourselves. Instead of allocating memory for each node individually, we

allocate memory for all the nodes at once (i.e., an array of **structs**) and linearize the tree in the following way: the root lies at index 0. Assuming a perfectly-balanced tree, for a node at index i , its left child is at $2i + 1$ and its right child is at $2i + 2$. This is equivalent to laying out the tree using a breadth-first traversal of the nodes. The hope is that by manually controlling memory layout, we can achieve better reference locality, thereby speeding up the memory references. This is similar to the idea behind **CSS-Trees** [17] used in the database community. For convenience we call this the **STRUCT⁺** implementation.

One nice property of retaining the left and right pointers in this implementation is that for unbalanced trees (i.e., trees with missing nodes), we can more tightly pack the nodes to remove “empty space” (still following the layout approach based on breadth-first node traversal). Thus, the **STRUCT⁺** implementation occupies the same amount of memory as **STRUCT**, except that the memory is contiguous.

PRED: The **STRUCT⁺** implementation tackles the reference locality problem, but there remains one more issue: the presence of branches (resulting from the conditionals), which can be quite expensive to execute. Branch mispredicts may cause pipeline stalls and wasted cycles (and of course, we would expect many mispredicts with trees). Although it is true that speculative execution renders the situation far more complex, removing branches may yield performance increases. A well-known trick in the compiler community for overcoming these issues is known as predication [2, 13]. The underlying idea is to convert control dependencies (hazards) into data dependencies (hazards), thus altogether avoiding jumps in the underlying assembly code.

Here is how predication is adapted for our case: We encode the tree as a **struct** array in C, `nd`, where `nd[i].fid` is the feature id to examine, and `nd[i].theta` is the threshold. We assume a fully-branching binary tree, with nodes laid out via breadth-first traversal (i.e., for a node at index i , its left child is at $2i + 1$ and its right child is at $2i + 2$). To make the prediction, we probe the array in the following manner:

```
i = (i < 1) + 1 + (v[nd[i].fid] >= nd[i].theta);
i = (i < 1) + 1 + (v[nd[i].fid] >= nd[i].theta);
...
```

We completely unroll the tree traversal loop, so the above statement is repeated d times for a tree of depth d . At the end, i contains the index of the leaf node corresponding to the prediction (which we look up in another array). One final implementation detail: we hard code a prediction function for each tree depth, and then dispatch dynamically using function pointers. Note that this approach assumes a fully-balanced binary tree; to cope with unbalanced trees, we expand by inserting dummy nodes.

VPRED: Predication eliminates branches but at the cost of introducing data hazards. Each statement in **PRED** requires an indirect memory reference. Subsequent instructions cannot execute until the contents of the memory locations are fetched—in other words, the processor will simply stall waiting for memory references to resolve. Therefore, predication is entirely bottlenecked on memory access latencies.

A common technique adopted in the database literature to mask these memory latencies is called *vectorization* [4, 24]. Applied to our task, this translates into operating on multiple instances (feature vectors) at once, in an interleaved way. This takes advantage of multiple dispatch and pipelining in

modern processors (provided that there are no dependencies between dispatched instructions, which is true in our case). So, while the processor is waiting for the memory access from the predication step on the first instance, it can start working on the second instance. In fact, we can work on v instances in parallel. For $v = 4$, this looks like the following, working on instances `i0`, `i1`, `i2`, `i3` in parallel:

```
i0 = (i0<<1) + 1 + (v[nd[i0].fid] >= nd[i0].theta);
i1 = (i1<<1) + 1 + (v[nd[i1].fid] >= nd[i1].theta);
i2 = (i2<<1) + 1 + (v[nd[i2].fid] >= nd[i2].theta);
i3 = (i3<<1) + 1 + (v[nd[i3].fid] >= nd[i3].theta);

i0 = (i0<<1) + 1 + (v[nd[i0].fid] >= nd[i0].theta);
i1 = (i1<<1) + 1 + (v[nd[i1].fid] >= nd[i1].theta);
i2 = (i2<<1) + 1 + (v[nd[i2].fid] >= nd[i2].theta);
i3 = (i3<<1) + 1 + (v[nd[i3].fid] >= nd[i3].theta);
...
```

In other words, we traverse one layer in the tree for four instances at once. While we’re waiting for `v[nd[i0].fid]` to resolve, we dispatch instructions for accessing `v[nd[i1].fid]`, and so on. Hopefully, by the time the final memory access has been dispatched, the contents of the first memory access are available, and we can continue without processor stalls.

Again, we completely unroll the tree traversal loop, so each block of statements is repeated d times for a tree of depth d . At the end, i contains the index of the leaf nodes corresponding to the prediction for v instances. Setting v to 1 reduces this model to pure predication (i.e., no vectorization). Note that the optimal value of v is dependent on the relationship between the amount of computation performed and memory latencies—we will determine this relationship empirically. For convenience, we refer to the vectorized version of the predication technique as VPRED.

4. EXPERIMENTAL SETUP

Given that the focus of our work is efficiency, our primary evaluation metric is prediction speed. We define this as the elapsed time between the moment a feature vector (i.e., a test instance) is presented to the tree-based model to the moment that a prediction (in our case, a regression value) is made for the instance. To increase the reliability of our results, we conducted multiple trials and report the mean and variance.

We conducted two sets of experiments: first, using synthetically-generated data to quantify the performance of individual trees in isolation, and second, on standard learning-to-rank datasets to verify the performance of full ensembles.

All experiments were run on a Red Hat Linux server, with Intel Xeon Westmere quad-core processors (E5620 2.4GHz). This architecture has a 64KB L1 cache per core, split between data and instructions; a 256KB L2 cache per core; and a 12MB L3 cache shared by all cores. Code was compiled with GCC (version 4.1.2) using optimization flags `-O3 -fomit-frame-pointer -pipe`. All code ran single-threaded.

4.1 Synthetic Data

The synthetic data consisted of randomly generated trees and randomly generated feature vectors. Each intermediate node in a tree has two fields: a feature id and a threshold on which the decision is made. Each leaf is associated with a regression value. Construction of a random tree of depth d begins with the root node. We pick a feature id at random and generate a random threshold to split the tree into left and right subtrees. This process is recursively performed

to build each subtree until we reach the desired tree depth. When we reach a leaf node, we generate a regression value at random. Note that our randomly-generated trees are fully-balanced, i.e., a tree of depth d has 2^d leaf nodes.

Once a tree has been constructed, the next step is to generate random feature vectors. Each random feature vector is simply a floating-point array of length f (= number of features), where each index position corresponds to a feature value. We assume that all paths in the decision tree are equally likely; the feature vectors are generated in a way that guarantees an equal likelihood of visiting each leaf. To accomplish this, we take one leaf at a time and follow its parents back to the root. At each node, we take the node’s feature id and produce a feature value based on the position of the child node. That is, if the child node we have just visited is on the left subtree we generate a feature value that is smaller than the threshold stored at the current parent node; otherwise we generate a feature value larger than the threshold. We randomize the order of instances once we have generated all the feature vectors. To avoid any cache effects, our experiments are conducted on a large number of instances (512k).

Given a random tree and a set of random feature vectors, we ran experiments to assess the various implementations of tree-based models described in Section 3. To get a better sense of the variance, we performed 5 trials; in each trial we constructed a new random binary tree and a different randomly-generated set of feature vectors. To explore the design space, we conducted experiments with varying tree depths $d \in \{3, 5, 7, 9, 11\}$ and varying feature sizes $f \in \{32, 128, 512\}$.

4.2 Learning-to-Rank Experiments

In addition to randomly-generated trees, we conducted experiments using standard learning-to-rank datasets, where training, validation, and test data are provided. Using the training and validation sets we learned a complete tree-ensemble ranking model, and evaluation is then carried out on test instances to determine the speed of the various implementations. These experiments assess performance in a real-world application.

We used gradient-boosted regression trees (GBRTs) [5, 21, 8] to train a learning-to-rank model. GBRTs are ensembles of regression trees that yield state-of-the-art effectiveness on learning-to-rank tasks. The learning algorithm sequentially adds new trees to the ensemble that best account for the remaining regression error (i.e., the residuals). We used the open-source `jforests` implementation⁵ of LambdaMART to optimize NDCG [11]. Although there is no way to precisely control the depth of each tree, we can adjust the size distribution of the trees by setting a cap on the number of leaves (which is an input parameter to the learner).

We used two standard learning-to-rank datasets: LETOR-MQ2007⁶ and MSLR-WEB10K.⁷ Both are pre-folded, providing training, validation, and test instances. Table 1 shows the dataset sizes and the numbers of features. To measure variance, we repeated experiments on all five folds. Note that MQ2007 is much smaller and is considered by many in the community to be outdated.

The values of f (number of features) in our synthetic ex-

⁵<http://code.google.com/p/jforests/>

⁶<http://research.microsoft.com/en>

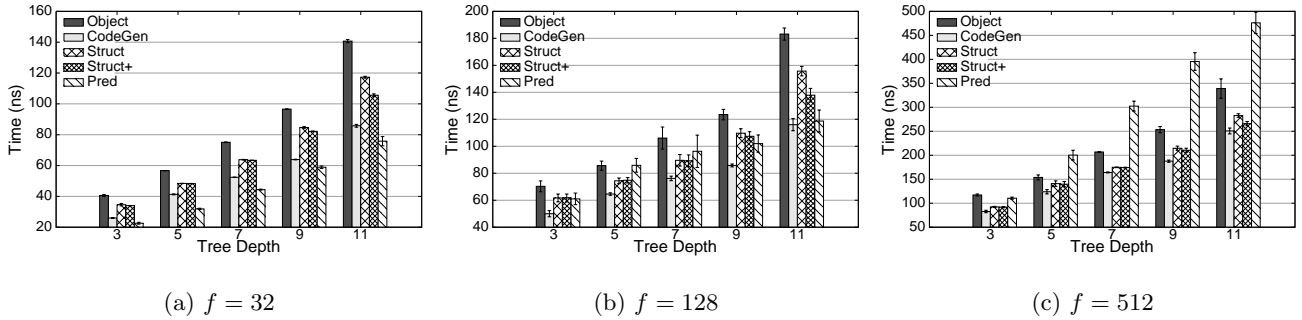


Figure 1: Prediction time per instance (in nanoseconds) on synthetic data using various implementations.

Table 1: Average number of training, validation, and test instances in our learning-to-rank datasets, along with the number of features.

| Dataset | Train | Validate | Test | Features |
|--------------|-------|----------|------|----------|
| MSLR-WEB10K | 720K | 240K | 240K | 136 |
| LETOR-MQ2007 | 42K | 14K | 14K | 46 |

periments are guided by these learning-to-rank datasets. We selected feature sizes that are multiples of 16 (4-byte floats) so that the feature vectors are integer multiples of cache line sizes (64 bytes): $f = 32$ roughly corresponds to LETOR features and is representative of a small feature space; $f = 128$ corresponds to MSLR and is representative of a medium-sized feature space. We introduced a third condition $f = 512$ to capture a large feature space condition.

5. RESULTS

In this section we present experimental results, beginning with evaluation on synthetic data and then on learning-to-rank datasets.

5.1 Synthetic Data: Base Results

We begin by focusing on the first five implementations described in Section 3 (leaving aside VPRED for now), using the procedure described in Section 4.1. The prediction time per randomly-generated test instance is shown in Figure 1, measured in nanoseconds. The balanced randomly-generated trees vary in terms of tree depth d , and each bar chart shows a separate value of f (number of features). Time is averaged across five trials and error bars denote 95% confidence intervals. It is clear that as trees become deeper, prediction speeds decrease overall. This is obvious since deeper trees require more feature accesses and predicate checks, more pointer chasing, and more branching (depending on the implementation).

First, consider the high-flexibility and high-performance baselines. As expected, the OBJECT implementation is the slowest (except for PRED with $f = 512$). It is no surprise that the C++ implementation is slow due to the overhead from classes and objects (recall the other implementations are in C). The gap between OBJECT and STRUCT, which is the comparable C implementation, grows with larger trees.

Also as expected, the CODEGEN implementation is very fast: with the exception of $f = 32$, hard-coded if-else statements are faster or just as fast as all other implementations, regardless of tree depth.

Comparing STRUCT⁺ with STRUCT, we observe no significant improvement for shallow trees, but a significant speedup for deep trees. Recall that in STRUCT⁺, we allocate memory for the entire tree so that it resides in a contiguous memory block, whereas in STRUCT we let malloc allocate memory however it chooses. This shows that reference locality is important for deeper trees.

Finally, turning to the PRED condition, we observe a very interesting behavior. For small feature vectors $f = 32$, the technique is actually faster than CODEGEN. This shows that for small feature sizes, predication helps to overcome branch mispredicts, i.e., converting control dependencies into data dependencies increases performance. For $f = 128$, results are mixed compared to CODEGEN, STRUCT, and STRUCT⁺: sometimes faster, sometimes slower. However, for large feature vectors ($f = 512$), the performance of PRED is terrible, even worse than the OBJECT implementation. We explain this result as follows: PRED performance is entirely dependent on memory latency. When traversing the tree, it needs to wait for the contents of memory before proceeding. Until the memory reference is resolved, the processor simply stalls. With small feature vectors, we get excellent locality: 32 features take up two 64-byte cache lines, which means that evaluation incurs at most two cache misses. Since memory is fetched by cache lines, once a feature is accessed, accesses to all other features on the same cache line are essentially “free”. Locality decreases as the feature vector size increases: the probability that the predicate at a tree node accesses a feature close to one that has already been accessed goes down. Thus, as the feature vector size grows, the PRED prediction time becomes increasingly dominated by stalls waiting for memory fetches.

The effect of this “memory wall” is evident in the other implementations as well. We observe that the performance differences between CODEGEN, STRUCT, and STRUCT⁺ shrink as the feature size increases (whereas they are more pronounced for smaller feature vectors). This is because as feature vector size increases, more and more of the prediction time is dominated by memory latencies.

How can we overcome these memory latencies? Instead of simply stalling while we wait for memory references to resolve, we can try to do other useful computation—this is exactly what vectorization is designed to accomplish.

us/um/beijing/projects/letor/letor4dataset.aspx

⁷<http://research.microsoft.com/en-us/projects/mslr/>

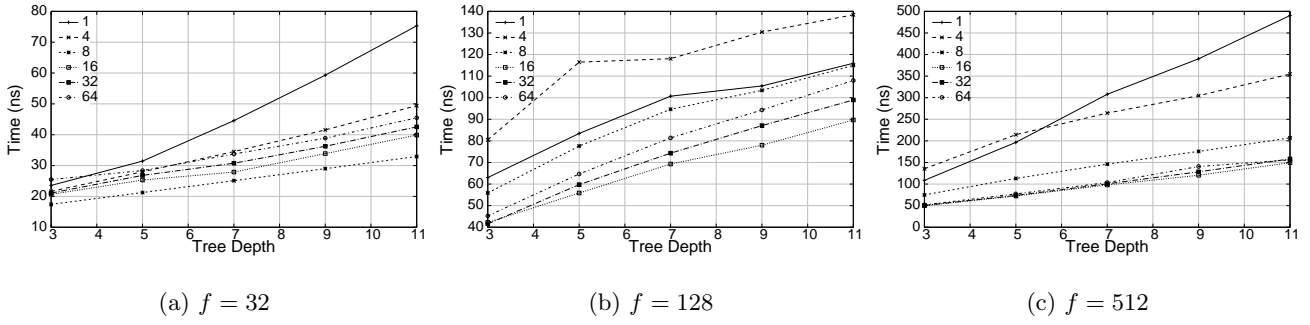


Figure 2: Prediction time per instance (in nanoseconds) on synthetic data using vectorized predication, for varying values of the batch size v .

5.2 Tuning Vectorization Parameter

In Section 3, we proposed *vectorization* of the predication technique in order to mask memory latencies. The idea is to work on v instances (feature vectors) at the same time, so that while the processor is waiting for memory access for one instance, useful computation can happen on another. This takes advantage of pipelining and multiple dispatch in modern superscalar processors.

The effectiveness of vectorization depends on the relationship between time spent in actual computation and memory latencies. For example, if memory fetches take only one clock cycle, then vectorization cannot possibly help. The longer the memory latencies, the more we would expect vectorization (larger batch sizes) to help. However, beyond a certain point, once memory latencies are effectively masked by vectorization, we would expect larger values of v to have little impact. In fact, values that are too large start to bottleneck on memory bandwidth and cache size.

In Figure 2, we show the impact of various batch sizes, $v \in \{1, 4, 8, 16, 32, 64\}$, for the different feature sizes. Note that when v is set to 1, we evaluate one instance at a time, which reduces to the PRED implementation. Prediction speed is measured in nanoseconds and normalized by batch size (i.e., divided by v), so we report *per-instance* prediction time. For $f = 32$, $v = 8$ yields the best performance; for $f = 128$, $v = 16$ yields the best performance; for $f = 512$, $v = \{16, 32, 64\}$ all provide approximately the same level of performance. These results are exactly what we would expect: since memory latencies increase with larger feature sizes, a larger batch size is needed to mask the latencies.

With the combination of vectorization and predication, VPRED becomes the fastest of all our implementations on the synthetic data. Comparing Figures 1 and 2, we see that VPRED (with optimal vectorization parameter) is actually faster than CODEGEN. Table 2 summarizes this comparison. Vectorization is up to 70% faster than the non-vectorized implementation; VPRED can be twice as fast as CODEGEN. In other words, we retain the best of both words: speed and flexibility, since the VPRED implementation does not require code recompilation.

5.3 Learning-to-Rank Experiments

Having evaluated different implementations on synthetic data, we move on to learning-to-rank datasets using tree ensembles. As previously described, we used the implementation of LambdaMART by Ganjisaffar et al. [8]. Once a model has been trained and validated, we evaluate on the

Table 2: Prediction time per instance (in nanoseconds) for the vectorized predication implementation, compared to simple predication and code generation, along with relative improvements.

(a) $f = 32, v = 8$

| d | VPRED | PRED | Δ | CODEGEN | Δ |
|-----|-------|------|----------|---------|----------|
| 3 | 17.4 | 22.6 | 23% | 26.0 | 33% |
| 5 | 21.3 | 31.9 | 33% | 41.3 | 48% |
| 7 | 25.1 | 44.4 | 44% | 52.4 | 52% |
| 9 | 28.9 | 58.9 | 51% | 63.9 | 55% |
| 11 | 39.2 | 75.8 | 57% | 85.8 | 54% |

(b) $f = 128, v = 16$

| d | VPRED | PRED | Δ | CODEGEN | Δ |
|-----|-------|-------|----------|---------|----------|
| 3 | 42.2 | 61.0 | 31% | 50.0 | 16% |
| 5 | 55.8 | 85.9 | 35% | 64.6 | 14% |
| 7 | 69.3 | 96.3 | 28% | 76.2 | 9% |
| 9 | 77.9 | 102.0 | 24% | 85.8 | 9% |
| 11 | 89.6 | 118.7 | 25% | 116.0 | 23% |

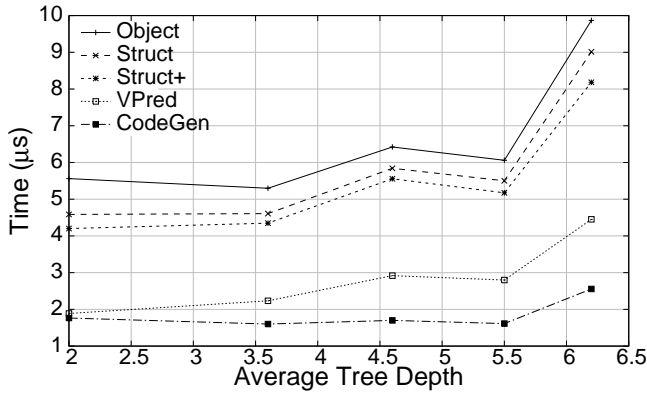
(c) $f = 512, v = 16$

| d | VPRED | PRED | Δ | CODEGEN | Δ |
|-----|-------|-------|----------|---------|----------|
| 3 | 49.7 | 110.6 | 55% | 82.8 | 40% |
| 5 | 72.3 | 200.3 | 64% | 123.9 | 42% |
| 7 | 97.8 | 302.5 | 68% | 164.2 | 40% |
| 9 | 120.4 | 395.5 | 70% | 187.8 | 36% |
| 11 | 149.5 | 476.1 | 69% | 250.7 | 40% |

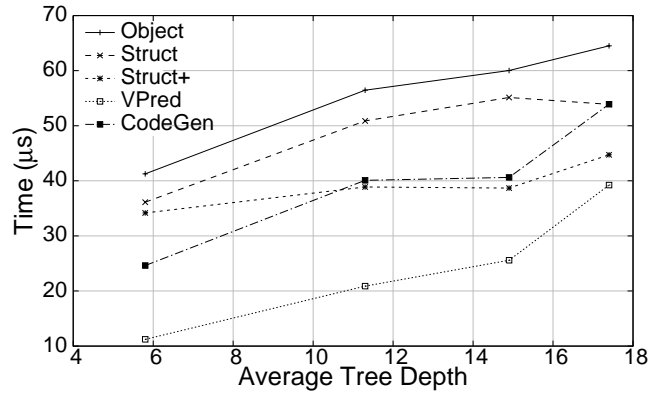
test set to measure prediction speed. Since the datasets come pre-folded five ways, we repeated our experiments five times and report mean and variance across the runs.

To handle ensembles in our implementations, we simply add an outer loop to the algorithm that iterates over individual trees in the ensemble. Note that Ganjisaffar et al. actually construct multiple ensembles, each built using a random bootstrap of the training data (i.e., *bagging* multiple boosted ensembles). In this work, we do not adopt this procedure because bagging is embarrassingly parallel from the runtime execution perspective and hence not particularly interesting. For learning parameters, we used values recommended by Ganjisaffar et al., with the exception of max leaves (see below). Feature and data sub-sampling parameters were set to 0.3, minimum percentage of observations per leaf was set to 0.5, and the learning rate was set to 0.05.

In terms of performance, shallower trees are naturally pre-



(a) LETOR-MQ2007



(b) MSLR-WEB10K

Figure 3: Per-instance prediction times (in microseconds), for ensembles of trees trained using LambdaMART on different datasets.

Table 3: NDCG and average tree depth (variance in parentheses) measured across five folds using various settings for max number of leaves. For MSLR, + and * show statistically significant improvements over models obtained by setting “Max. Leaves” to 10 and 30 respectively.

(a) LETOR-MQ2007

| Max. Leaves | Avg. Depth | @1 | @3 | @20 |
|-------------|------------|-------|-------|-------|
| 3 | 2.0 (0.0) | 0.469 | 0.476 | 0.590 |
| 5 | 3.6 (0.2) | 0.463 | 0.478 | 0.588 |
| 7 | 4.6 (0.6) | 0.490 | 0.484 | 0.592 |
| 9 | 5.5 (1.0) | 0.475 | 0.477 | 0.589 |
| 11 | 6.2 (1.2) | 0.478 | 0.481 | 0.591 |

(b) MSLR-WEB10K

| Max. Leaves | Avg. Depth | @1 | @3 | @20 |
|-------------|-------------|--------------------|--------------------|----------------------|
| 10 | 5.8 (1.1) | 0.466 | 0.452 | 0.505 |
| 30 | 11.3 (6.2) | 0.470 ⁺ | 0.456 ⁺ | 0.510 ⁺ |
| 50 | 14.9 (10.7) | 0.475 ⁺ | 0.459 ⁺ | 0.512 ⁺ * |
| 70 | 17.4 (12.9) | 0.466 | 0.453 | 0.510 ⁺ |

ferred. But what is the relationship between tree depth and ranking effectiveness? Tree depth with our particular training algorithm cannot be precisely controlled, but can be indirectly influenced by the maximum number of leaves on an individual tree (an input to the learner). Table 3 shows the average NDCG values (at different ranks) measured across five folds on the LETOR and MSLR datasets with different values of this parameter, similar to the range of values explored in [8]. Statistical significance was tested using the Wilcoxon test (p -value 0.05); none of the differences on the LETOR dataset were significant. For each condition, we also report the average depth of the trees that were actually learned. The average tree depth is computed for every ensemble and then averaged across the five folds; variance is presented in parentheses.

Results show that for LETOR, tree depth makes no significant difference on NDCG, whereas larger trees yield better results on MSLR; however, there appears to be little dif-

ference between 50 and 70 max leaves. The results make sense: to exploit larger feature spaces we need trees with more nodes. Since many in the community consider the LETOR dataset to be out of date with an impoverished feature set, more credence should be given to the MSLR results.

Turning to performance results, Figure 3 illustrates per-instance prediction speed for various implementations on the learning-to-rank datasets. Note that this is on the entire ensemble, with latencies now measured in microseconds instead of nanoseconds. As described above, the trees were trained with different settings of max leaves; the x -axis plots the tree depths from Table 3. In this set of experiments, we made use of the VPRED approach with the vectorization parameter set to 8 for LETOR and 16 for MSLR.

Results from the synthetic datasets mostly carry over to these learning-to-rank datasets. OBJECT is the slowest implementation and STRUCT is slightly faster. On the LETOR dataset, STRUCT is only slightly slower than STRUCT⁺, but on MSLR, STRUCT⁺ is faster than STRUCT by a larger margin in most cases. VPRED outperforms all other techniques, including CODEGEN on MSLR, but is slower than CODEGEN on LETOR (except for the shallowest trees). However, note that in terms of NDCG, Table 3(a) shows no difference in effectiveness, so there is no advantage to building deeper trees for LETOR.

The conclusion appears clear: for tree-based ensembles on real-world learning-to-rank datasets, we can achieve the best of both worlds. With a combination of predication and vectorization, we can make predictions faster than statically-generated if-else blocks, yet retain the flexibility in being able to specify the model dynamically, which enables rapid experimentation.

6. DISCUSSION AND FUTURE WORK

Our experiments show that predication and vectorization are effective techniques for substantially increasing the performance of tree-based models, but one potential objection might be: are we measuring the right thing? In our experiments, prediction time is measured from when the feature vector is presented to the model to when the prediction is made. Critically, we assume that features have already been computed. What about an alternative architecture where

Table 4: Average percentage of examined features (variance in parentheses) across five folds using various max-number-of-leaves settings.

(a) LETOR-MQ2007

| | 3 | 5 | 7 | 9 | 11 |
|------------------------|---------------|---------------|---------------|---------------|---------------|
| Percentage of features | 76.7 (5.0) | 72.2 (8.8) | 80.2 (5.6) | 77.6 (7.6) | 84.8 (1.9) |

(b) MSLR-WEB10K

| | 10 | 30 | 50 | 70 |
|------------------------|---------------|---------------|---------------|---------------|
| Percentage of features | 92.7 (1.7) | 96.5 (1.1) | 96.3 (1.9) | 95.6 (1.6) |

features are computed lazily, i.e., only when the predicate at a tree node needs to access a particular feature?

This alternative architecture, where features are computed on demand, is difficult to study since results will be highly dependent on the implementation of feature extraction—which in turn depends on the underlying data structures (layout of the inverted indexes), compression techniques, and how computation-intensive the features are. However, there is a much easier way to study this issue—we can trace the execution of the full tree ensemble and keep track of the fraction of features that are accessed. If during the course of making a prediction, most of the features are accessed, then there is little waste in computing all the features first and then presenting the complete feature vector to the model.

Table 4 shows the average fraction of features accessed in the final learned models for both learning-to-rank datasets, with different max leaves configurations. It is clear that, for both datasets, most of the features are accessed during the course of making a prediction, and in the case of the MSLR dataset, nearly all the features are accessed all the time (especially with deeper trees, which yield higher effectiveness). Therefore, it makes sense to separate feature extraction from prediction. In fact, there are independent compelling reasons to do so: a dedicated feature extraction stage can benefit from better reference locality (when it comes to document vectors, postings, or whatever underlying data structures are necessary for computing features). Interleaving feature extraction with tree traversal may lead to “cache churn”, where a particular data structure is repeatedly loaded and then displaced by other data.

Returning to a point in the introduction: do these optimizations actually matter, in the broader context of real-world search engines? This is of course a difficult question to answer and highly dependent on the actual search architecture, which is a complex distributed system spanning hundreds of machines or more. Here, we venture some rough estimates. From Figure 3(b), the MSLR dataset, we see that compared to CODEGEN, VPRED reduces per-instance prediction time from around $40\mu\text{s}$ to around $25\mu\text{s}$ (for max leaves setting of 50); this translates into a 38% reduction in latency per instance. In a web search engine, the learning to rank algorithm is applied to a candidate list of documents that is usually generated by other means (e.g., scoring with BM25 and a static prior). The exact details are proprietary, but the published literature does provide some clues. For example, Cambazoglu et al. [6] (authors from Yahoo!)

experimented with reranking 200 candidate documents to produce the final ranked list of 20 results (the first two pages of search results). From these numbers, we can compute the per-query reranking time to be 8ms using the CODEGEN approach and 5ms with VPRED. This translates into an increase from 125 queries per second to 200 queries per second on a single thread for this phase of the search pipeline. Alternatively, gains from faster prediction can be leveraged to rerank more results or take advantage of more features. This simple estimate suggests that our optimizations can make a noticeable difference in web search, and given that our techniques are relatively simple—the predication and vectorization optimizations definitely seem worthwhile.

During the course of our experiments, we noticed that two assumptions of our implementations did not appear to be fully valid. First, the PRED and VPRED implementations assume fully-balanced binary trees (i.e., every node has a left and a right child). In contrast, recall that STRUCT⁺ makes no such assumption because with the left and right pointers we can tightly pack the tree nodes. The fully-balanced tree assumption does not turn out to be valid for GBRTs—the learner does not have a preference for any particular tree topology, and so the trees are unbalanced most of the time. To compensate for this, the PRED and VPRED implementations require insertion of dummy nodes to create a fully-balanced tree. Second, we assume that all paths are equally likely in a tree, i.e., that at each node, the left and right branches are taken with roughly-equal frequency. We noticed, however, that this is often not the case. To the extent that one branch is favored over another, branch prediction provides non-predicated implementations (i.e., if-else blocks) an advantage, since branch prediction will guess correctly more often, thus avoiding pipeline flushes.

One promising future direction to address the above two issues is to adapt the model learning process to prefer balanced trees and predicates that divide up the feature space evenly. We believe this can be incorporated into the learning algorithm as a penalty, much in the same way that regularization is performed on the objective in standard machine learning. Thus, it is perhaps possible to jointly learn models that are both fast and good, as in the recently-proposed “learning to *efficiently* rank” framework [22, 23].

7. CONCLUSION

Modern processor architectures are incredibly complex because technological improvements have been uneven. This paper focuses on one particular issue: not all memory references are equally fast, and in fact, latency can differ by an order of magnitude. There are a number of mechanisms to mask these latencies, although it largely depends on developers knowing how to exploit these mechanisms. The database community has been exploring these issues for quite some time now, and in this respect the information retrieval, machine learning, and data mining communities are behind.

In this paper, we demonstrate that two relatively simple techniques, predication and vectorization, can significantly accelerate prediction performance for tree-based models, both on synthetic data and on real-world learning-to-rank datasets. To our knowledge, this is the first study of architecture-conscious implementations for a machine learning application—but we believe there are plenty of similar opportunities in other areas of machine learning as well.

8. ACKNOWLEDGMENTS

This work has been supported by NSF under awards IIS-0916043, IIS-1144034, and IIS-1218043. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsor. The first author's deepest gratitude goes to Katherine, for her invaluable encouragement and wholehearted support. The second author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob.

9. REFERENCES

- [1] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a modern processor: Where does time go? *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB1999)*, pp. 266–277, Edinburgh, Scotland, 1999.
- [2] D. August, W. Hwu, and S. Mahlke. A framework for balancing control flow and predication. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO1997)*, pp. 92–103, Research Triangle Park, North Carolina, 1997.
- [3] P. Boncz, M. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.
- [4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR2005)*, Asilomar, California, 2005.
- [5] C. Burges. From RankNet to LambdaRank to LambdaMART: An overview. Technical Report MSR-TR-2010-82, Microsoft Research, 2010.
- [6] B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM 2010)*, pp. 411–420, New York, 2010.
- [7] A. Criminisi, J. Shotton, and E. Konukoglu. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends in Computer Graphics and Vision*, 7(2–3):81–227, 2011.
- [8] Y. Ganjisaffar, R. Caruana, and C. Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR2011)*, pp. 85–94, Beijing, China, 2011.
- [9] G. Ge and G. Wong. Classification of premalignant pancreatic cancer mass-spectrometry data using decision tree ensembles. *BMC Bioinformatics*, 9:275, 2008.
- [10] B. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Morgan & Claypool Publishers, 2009.
- [11] K. Järvelin and J. Kekäläinen. Cumulative gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.
- [12] N. Johnson, G. Zhao, E. Hunsader, J. Meng, A. Ravindar, S. Carran, and B. Tivnan. Financial black swans driven by ultrafast machine ecology. *arXiv:1202.1448v1*, 2012.
- [13] H. Kim, O. Mutlu, Y. Patt, and J. Stark. Wish branches: Enabling adaptive and aggressive predicated execution. *IEEE Micro*, 26(1):48–58, 2006.
- [14] H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool Publishers, 2011.
- [15] K. Olukotun and L. Hammond. The future of microprocessors. *ACM Queue*, 3(7):27–34, 2005.
- [16] B. Panda, J. Herbach, S. Basu, and R. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB2009)*, pp. 1426–1437, Lyon, France, 2009.
- [17] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB1999)*, pp. 78–89, Edinburgh, Scotland, 1999.
- [18] K. Ross, J. Cieslewicz, J. Rao, and J. Zhou. Architecture sensitive database design: Examples from the Columbia group. *Bulletin of the Technical Committee on Data Engineering*, 28(2):5–10, 2005.
- [19] L. Schietgat, C. Vens, J. Struyf, H. Blockeel, D. Kocev, and S. Džeroski. Predicting gene function using hierarchical multi-label decision tree ensembles. *BMC Bioinformatics*, 11:2, 2010.
- [20] K. Svore and C. Burges. Large-scale learning to rank using boosted decision trees. *Scaling Up Machine Learning*. Cambridge University Press, 2011.
- [21] S. Tyree, K. Weinberger, and K. Agrawal. Parallel boosted regression trees for web search ranking. *Proceedings of the 20th International Conference on World Wide Web (WWW2011)*, pp. 387–396, Hyderabad, India, 2011.
- [22] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR2011)*, pp. 105–114, Beijing, China, 2011.
- [23] Z. Xu, K. Weinberger, and O. Chapelle. The greedy miser: Learning under test-time budgets. *Proceedings of the 29th International Conference on Machine Learning (ICML 2012)*, Edinburgh, Scotland, 2012.
- [24] M. Zukowski, P. Boncz, N. Nes, and S. Héman. MonetDB/X100—a DBMS in the CPU cache. *Bulletin of the Technical Committee on Data Engineering*, 28(2):17–22, 2005.